A Variation of Knoop, Rüthing, and Steffen's *Lazy Code Motion*

by Karl-Heinz Drechsler and Manfred P. Stadel
Siemens Nixdorf Informationssysteme AG, STM SD 2
Otto Hahn Ring 6, 8000 München 83, Germany

Morel and Renvoise discussed in 1979 a universal method for global code optimization in a compiler. By suppressing partial redundancies their method achieves several optimization techniques - such as moving loop invariant computations out of a loop or deleting redundant computations - at once. Their algorithm moves computations to earlier places in execution paths to eliminate partial redundancies. The movement is controlled by some heuristics.
The heuristics and algorithms used by Morel and Renvoise sometimes cause difficulties for practical use. Subsequent papers partly circumvented these difficulties by slightly modifying the heuristics and the algorithms. Knoop, Rüthing, and Steffen published in their paper *Lazy Code Motion* an optimal algorithm for the elimination of partial redundancies which entirely prevents these difficulties. This paper presents a variant of their algorithm which is better prepared for practical use.

1. INTRODUCTION

Morel and Renvoise discussed in their paper [5] a universal method for global code optimization in a compiler. By suppressing partial redundancies their method achieves several optimization techniques - such as moving loop invariant computations out of a loop or deleting redundant computations - at once. The algorithm is based on a bi-directional data flow analysis. It moves computations to earlier places in execution paths and thereby eliminates partial redundancies. The movement is controlled by some heuristics which are mainly due to a term named CONST in [5].

The heuristics and algorithms used in Morel and Renvoise's original paper [5] sometimes cause difficulties in practical use. Subsequent papers partly circumvented these difficulties by modifications of the heuristics and the data flow equations to be solved. In [4] Knoop, Rüthing, and Steffen published an optimal algorithm for the elimination of partial redundancies which entirely prevents these difficulties. Their algorithm is based on flow graphs with single statements as nodes. From a practical point of view flow graphs with entire basic blocks as nodes would be preferable.

Section 3 of this paper presents a variant of the idea in [4] which is based on flow graphs with entire basic blocks rather than single instructions as nodes. Further, in contrast to [4] and similar to [3], the computational model of a multi-register machine (with an arbitrary number of pseudo registers) is assumed. This allows to perform code motion after instruction selection and makes a special treatment of temporaries superfluous (saving one of the systems of data flow equations). Similar as the algorithms in [4] the algorithms presented in this paper entirely prevent the difficulties mentioned in [1], [2], and [3]. The idea from [4] of eliminating partial redundancies by moving computations to earlier places is kept, but a different data flow frame work is used to achieve this. Therefore, correctness and also optimality of the new algorithms need to be proved again which is done in section 4. Practical results are given in section 5.

Moving a computation to an earlier place in an execution path increases the live range of registers holding result values of the computations. Therefore, some of the difficulties with Morel and Renvoise's original method are unnecessary (redundant) code movements. This problem was addressed in several papers [1], [2], [3]. They improved the behaviour of code movement but were still not able to entirely prevent redundant code movements.

Since Morel and Renvoise only inserted new computations at the end of basic blocks, they could not eliminate all partial redundancies. To allow the elimination of more partial redundancies [1], [2], [3], and [4] took edge placements into consideration (or assumed that *critical edges* have been prevented by inserting empty blocks where necessary). The possibility of edge placement was already mentioned by Morel and Renvoise but not used in their original paper [5].

Drechsler and Stadel discussed in [3] a solution to prevent the difficulty of the movement of complex expressions to earlier places in execution paths than some of their subexpressions.

In general, solving a bi-directional system of data flow equations like that given in [5] is less efficient than solving a uni-directional system. By using edge placement Drechsler and Stadel [3] could reduce the data flow problem to a uni-directional system of equations. The systems of equations proposed by Knoop, Rüthing, and Steffen in [4] are also all uni-directional. The systems of equations suggested by Dhamdhere in [1] and [2] are *weak bi-directional* and therefore have the same low computational complexity as uni-directinal ones.

The method presented in this paper uses the same basic idea of two phases as Knoop, Rüthing, and Steffen in [4]. As in [4], the first phase determines the earliest possible placements which allow the elimination of as many partial redundancies as possible. In the second phase these placements are moved forward to the latest possible places without affecting the elimination of partial redundancies. However, the use of the information already contained in the previously computed values for availability (AVOUT) and anticipability (ANTIN) leads to simpler and fewer data flow equations.

2. NOTATIONS AND DEFINITIONS

The same notations as in [5] are used throughout this paper. They are now briefly summarized. It is assumed that the program is represented as a flow graph with a single entry node *entry*, from which all other nodes can be reached, and a single exit node *exit*. The nodes of the flow graph are basic blocks. A basic block is a sequence of primitive commands with a single entry point (usually a label) and a single exit point (usually a jump, branch, or switch command). In the following basic blocks are named by letters *i, j*, ... An edge from a block *i* to a block *j* in the flow graph is denoted by (i, j); *i* is then a *predecessor* of *j*, *j* a *successor* of *i*. The set of all predecessors of *j* is denoted by PRED_j, the set of all successors of *i* by SUCC_i. We assume that $\text{PRED}_{entry} = \text{SUCC}_{exit} = \emptyset$. Greek letters are used to identify paths: a path $\pi = \pi[i, j]$ from a node *i* to another node *j* is a sequence of nodes linked by edges and starting with *i* and ending with *j*. We sometimes also use the notation $\pi]i, j[$ if we want to exclude end nodes *i* or *j* from the path. A subpath of π from a node *n* on π to another node *m* on π is denoted by $\pi[n, m]$.

As computational model for the commands a multi-register machine with an arbitrary number of (pseudo-) registers is used. It is assumed that commands computing expressions (*computations*) deliver their results in pseudo-registers, and that all computations of the same expression always use the same pseudo-register as the result register. When the result of an expression is used as operand in another command, then its uniquely associated result register is taken. These assumptions simplify the elimination of redundant computations since it is not necessary to take care to save the result of a computation for later use as it is done in [4].

For each expression and each basic block Boolean properties are defined. In Boolean formulas we use the symbol $\cdot$ for the AND-operation, + for OR, and $\overline{}$ for NOT.

Definition of Local Properties

A basic block *i* is *transparent* for an expression *e*, $\text{TRANSP}_i\,(e)$, if none of *e*'s operands are modified (i.e. assigned new values) by commands in block *i*.

An expression *e* is *computed* in a basic block *i*, $\text{COMP}_i\,(e)$, if block *i* contains at least one computation of *e* and after the last computation of *e* in block *i* none of *e*'s operands are modified.

An expression *e* is *locally anticipatable* in a basic block *i*, $\text{ANTLOC}_i\,(e)$, if block *i* contains at least one computation of *e* and before the first computation of *e* in block *i* none of *e*'s operands are modified.

Definition of Global Properties

A path π in a flow graph is *transparent* for an expression e, $\text{TRANSP}_\pi(e)$, if all basic blocks k on π are transparent for e, i.e. $\text{TRANSP}_k(e)$.

An expression e is *available* at entry of block i, $\text{AVIN}_i(e)$, if every path $\pi[entry, i[$ contains at least one computation of e, i.e. $\text{COMP}_k(e) = \text{TRUE}$ for some k on $\pi[entry, i[$, and $\text{TRANSP}_{\pi]k,\,i[}(e) = \text{TRUE}$. Availability at exit of block i is then defined by

$$\text{AVOUT}_i(e) = \text{AVIN}_i(e) \cdot \text{TRANSP}_i(e) + \text{COMP}_i(e).$$

Availability of an expression e can be computed (iteratively) as the maximal solution of the system of equations

$$\begin{aligned} \text{AVIN}_i(e) &= \text{FALSE}, && \text{if } i \text{ is the entry block} \\ &= \prod_{j \in \text{PRED}_i} \text{AVOUT}_j(e), && \text{otherwise.} \end{aligned}$$

An expression e is *anticipatable* at exit of block i, $\text{ANTOUT}_i(e)$, if every path $\pi[i, exit]$ contains at least one computation of e, i.e. $\text{ANTLOC}_k(e) = \text{TRUE}$ for some k on $\pi]i, exit]$, and $\text{TRANSP}_{\pi]i,\,k[}(e) = \text{TRUE}$. Anticipability at entry of block i is then defined by

$$\text{ANTIN}_i(e) = \text{ANTOUT}_i(e) \cdot \text{TRANSP}_i(e) + \text{ANTLOC}_i(e).$$

Anticipability of an expression can (iteratively) be computed as maximal solution of the system of equations

$$\begin{aligned} \text{ANTOUT}_i(e) &= \text{FALSE}, && \text{if i is an exit block} \\ &= \prod_{j \in \text{SUCC}_i} \text{ANTIN}_j(e), && \text{otherwise.} \end{aligned}$$

All these computations can be done concurrently for all expressions; Boolean operations $\cdot$, $+$, and $\overline{}$ then become operations on bit-vectors. For the sake of clarity in the following we will often consider only one expression and thus drop argument (e) from Boolean properties.

The following additional definition is helpful for proofs of theorems.

Definition

A path $\pi[i,j[$ is said to *carry availability* (for an expression e) if there is a basic block k on π such that $\text{COMP}_k = \text{TRUE}$ and $\text{TRANSP}_{\pi]k,\,j[} = \text{TRUE}$.

Note that e is available at entry of j if and only if all paths $\pi[entry, j[$ carry availability. e is partially available at entry of j if and only if a path $\pi[entry, j[$ exists which carries availability. The concept of partial availability was introduced in [5], but this paper does not explicitly make use of it.

3. NEW SYSTEMS OF EQUATIONS FOR CODE MOTION

As in [5] it is now assumed that availability and anticipability have already been computed in a preliminary phase. In contrast to [5], however, the computation of partial availability is not required. As in [3] not only Boolean properties for the nodes of the flow graph are defined but also properties for edges.

In a first phase for each edge (i, j) of the flow graph a property $\mathrm{EARLIEST}_{i,j}$ is computed as follows:

$$\begin{aligned}\mathrm{EARLIEST}_{i,j} &= \mathrm{ANTIN}_j \cdot \overline{\mathrm{AVOUT}_i}, && \text{if } i \text{ is the entry block}\\ &= \mathrm{ANTIN}_j \cdot \overline{\mathrm{AVOUT}_i} \cdot (\overline{\mathrm{TRANSP}_i} + \overline{\mathrm{ANTOUT}_i}), && \text{otherwise}\end{aligned}$$

Note that - in contrast to similar equations in [4] - this is a local property of an edge rather than a global system of equations which would need to be solved iteratively. All the necessary global data flow information has already been collected during the previous computation of AVOUT and ANTIN.

In other words, for each path π reaching a computation of some expression e, $\mathrm{EARLIEST}_{i,j}$ = TRUE for the earliest edge on π where e is not available but where a computation of e could safely be inserted.As in [4], in a second phase a maximal solution of the following system of equations is iteratively computed

$$\begin{aligned}\mathrm{LATERIN}_j &= \mathrm{FALSE}, && \text{if } j \text{ is the entry block}\\ &= \prod_{i \in \mathrm{PRED}_j} \mathrm{LATER}_{i,j}, && \text{otherwise}\\ \mathrm{LATER}_{i,j} &= \mathrm{LATERIN}_i \cdot \overline{\mathrm{ANTLOC}_i} + \mathrm{EARLIEST}_{i,j}\end{aligned}$$

This is a uni-directional system which can efficiently be solved. In other words, LATER determines whether insertions of a computation can also be done later in the flow graph with the same effect. There is another intuitive explanation of LATER which is based on the following definition:

Definition

A path $\pi[i,j[$ is said to *carry (earliest) placeability* (for an expression e) if there is an edge (k, k') on π such that $\mathrm{EARLIEST}_{k,k'}$ = TRUE and $\mathrm{TRANSP}_{\pi[k',j[}$ = TRUE and there is no computation (of e) on $\pi[k',j[$.

Lemma 1 below shows that $\mathrm{LATERIN}_j$ = TRUE if and only if all paths reaching j carry placeability. In this sense $\mathrm{LATERIN}_j$ is similar to AVIN_j.

We are considering the insertion of computations in edges rather than at the end of basic blocks. The computation (of an expression e) is inserted in an edge (i, j) if $\mathrm{INSERT}_{i,j}$ = TRUE, where:

$$\mathrm{INSERT}_{i,j} = \mathrm{LATER}_{i,j} \cdot \overline{\mathrm{LATERIN}_j}$$

The first computation (of an expression e) in a block i is deleted if DELETE_i = TRUE, where:

$$\begin{aligned}\mathrm{DELETE}_i &= \mathrm{FALSE}, && \text{if } i \text{ is the entry block}\\ &= \mathrm{ANTLOC}_i \cdot \overline{\mathrm{LATERIN}_i}, && \text{otherwise}\end{aligned}$$

All together only three global and uni-directional systems of Boolean equations need to be solved iteratively: the systems for AVAIL, ANT, and LATER. When the original method of Morel and Renvoise [5] is applied three uni-directional systems of equations for AVAIL, PAVAIL (*partial availability*), and ANT, and in addition the bi-directinal system for PP (*placement possible*) have to be solved. The method of Knoop, Rüthing, and Steffen [4] requires the solution of 4 uni-directinal systems of equations.

4. THEORETICAL RESULTS

The correctness and optimality of the code motion method presented in the previous section will now be proved. Although these proofs use similar ideas as those in [4], they were developped independly.

LEMMA 1.

LATERIN$_j$ = TRUE if and only if all paths $\pi[entry, j[$ reaching block j carry placeability. LATER$_{i,j}$ = TRUE if and only if all paths $\pi[entry, i]$ carry placeability.

PROOF. LATERIN$_j$ = TRUE implies that every path $\pi[entry, j]$ reaching block j contains some edge (i, i') with EARLIEST$_{i,i'}$ = TRUE and that ANTLOC$_k$ = FALSE for all blocks k on $\pi[i', j[$, i.e. that there is no computation on $\pi[i', j[$. On the other hand, EARLIEST$_{i,i'}$ = TRUE implies ANTIN$_{i'}$ = TRUE, i.e. every path $[i', exit]$ contains a computation, say in block k, and the subpath to the first computation on that path is transparent. From these facts we conclude TRANSP$_{\pi[i', j[}$ = TRUE and thus π carries placeability. The case LATER$_{i,j}$ = TRUE is treated similarly.

Assume now that all paths reaching j carry placeability. It is now a direct consequence of the definitions that a maximal solution of the system of equations for LATER and LATERIN delivers LATERIN$_j$ = TRUE. ▯

LEMMA 2.

Every path $\pi = \pi[entry, i[$ reaching a computation in a block i (i.e. ANTLOC$_i$ = TRUE) carries availability or placeability.

If every path reaching a block i carries availability (i.e. AVIN$_i$ = TRUE) then none of them carries placeability.

PROOF. Consider a maximum transparent subpath $\pi]j, i[$, or $\pi[entry, i[$ in case of π is entirely transparent. If either COMP$_j$ = TRUE or there is some computation on $\pi]j, i[$ then π carries availability. Otherwise AVOUT$_j$ = FALSE for every block on $\pi[j, i[$.

Let j' be the successor of j on π (or the entry block in case of $\pi[entry, i[$ is entirely transparent). If ANTIN$_{j'}$ = TRUE then EARLIEST$_{j,j'}$ = TRUE too. Otherwise, since ANTIN$_i$ = TRUE and ANTIN$_{j'}$ = FALSE there must be an edge (k, k') on $\pi[j', i[$ such that ANTIN$_k$ $\cdot$ $\overline{\text{ANTIN}_{k'}}$ = TRUE. Then EARLIEST$_{k,k'}$ = TRUE and hence $\pi]j, i[$ carries placeability.

If all paths $\pi[entry, i[$ carry availability then everywhere after the last computations on these paths we have AVOUT$_k$ = TRUE and thus everywhere EARLIEST$_{k,k'}$ = FALSE. Therefore none of these paths can carry placeability. ▯

Note that a path may carry both, availability and placeability. An example is a path $\pi[1, 7[$ in the flow graph of Figure 1 below.

LEMMA 3.

A transparent path $\pi]i, j'[$ starting with an edge (i, i') and ending with an edge (j, j') such that EARLIEST$_{i,i'}$ and EARLIEST$_{j,j'}$ are both TRUE must contain some computation.

Similarily, a transparent path $\pi]i,j'[$ starting with an edge (i, i') and ending with an edge (j, j') such that $\text{INSERT}_{i,i'}$ and $\text{INSERT}_{j,j'}$ are both TRUE must also contain some computation.

PROOF. $\text{EARLIEST}_{j,j'}$ = TRUE and TRANSP_j = TRUE imply ANTOUT_j = FALSE. If there were no computation on π this would propagate to block i' such that $\text{ANTIN}_{i'}$ would be FALSE, a contradiction to $\text{EARLIEST}_{i,i'}$ = TRUE.

By lemma 1 $\text{INSERT}_{i,i'}$ = TRUE implies that every path reaching block i carries placeability, i.e. there is a transparent path starting with an edge (k, k'), ending at block i, and satisfying $\text{EARLIEST}_{k,k'}$ = TRUE. Further, there are no computations on that path between k' and i. A similar situation holds for (j, j'). Thus, the case for INSERT is reduced to the case for EARLIEST. □

THEOREM 1: CORRECTNESS.

Insertions are only done at places where the computation was anticipatable. Only those computations are deleted which would be redundant when all insertions have been done. After all insertions and deletions have been done no path contains more computations of an expression than before.

PROOF. Consider an edge (i, j) with $\text{INSERT}_{i,j}$ = TRUE. Lemma 1 states that every path π ending with edge (i, j) carries placeability, i.e. there is an edge (k, k') on π such that $\text{EARLIEST}_{k,k'}$ = TRUE and $\text{TRANSP}_{\pi[k',j[}$ = TRUE. $\text{EARLIEST}_{k,k'}$ = TRUE implies $\text{ANTIN}_{k'}$ = TRUE which would be impossible in the case of ANTIN_j = FALSE. Thus ANTIN_j = TRUE, which proves the first statement of the theorem.

By definition ANTIN_j = TRUE says that each path $\pi[j, \mathit{exit}]$ contains a computation, say in some block k on π (i.e. ANTLOC_k = TRUE), such that $\text{TRANSP}_{\pi[j,k[}$ = TRUE. Since we started our discussion from an edge (i, j) with $\text{INSERT}_{i,j}$ = TRUE we know that LATER_j = FALSE and this propagates to k such that we also have LATER_k = FALSE which implies DELETE_k = TRUE. This proves that each insertion is accompanied by a deletion on the same path.

Lemma 3 states that no two INSERTs reach the same computation.

It remains to be proved that all deletions are correct. Let k be a basic block with ANTLOC_k = TRUE. DELETE_k = TRUE then implies LATER_k = FALSE. Lemma 2 states that all paths reaching k either carry availability or placeability. From LATER_k = FALSE we conclude that all paths carrying placeability contain edges for which INSERT is TRUE. After these insertions have been done the first computation in block k is redundant. □

THEOREM 2: OPTIMALITY.

There is no placement causing less computations in some paths than the placement given by the values of INSERT and DELETE.

PROOF. We only need to show that remaining transparent paths from one computation to another cannot be prohibited by any other placement.

Assume the contrary. Then there is a transparent path $\pi]i,j[$ from one computation of an expression e in a basic block i (COMP_i = TRUE) to another in a block j (ANTLOC_j = TRUE). Let k' be the earliest block on $\pi]i,j[$ with $\text{ANTIN}_{k'}$ = TRUE. Let k be the predecessor of k' on π. Then either $k = i$ or ANTOUT_k = FALSE.

In case of AVOUT_k = FALSE, which excludes $k = i$, the only safe way to make the computation in j redundant would be to insert computations on $\pi[k',j]$. But then π would still be a transparent path from one computation (in i) to another (the inserted one). Thus, in this case, the partial redundancy in j cannot be eliminated by save insertions. An example of this situation is a path [1, 7] in the flow graph of Figure 1 below.

In case of AVOUT_k = TRUE all paths reaching k carry availability. Thus, by lemma 2, none of them can carry placeability, i.e. $\mathrm{LATERIN}_j$ = FALSE. If $j = k'$ we then have DELETE_j= TRUE. Otherwise, since $\mathrm{ANTIN}_{k'}$ = TRUE, TRANSP and ANTOUT are both TRUE everywhere on $\pi[k',j]$ which implies that EARLIEST must be FALSE for every edge on $\pi[k',j]$. This leads again to $\mathrm{LATERIN}_j$ = FALSE and hence DELETE_j = TRUE. $\square$

THEOREM 3.

Insertions are as late as possible as a maximal amount of partial redundancies are still being eliminated.

A consequence of Theorem 3 is that life ranges of pseudo registers are kept as short as possible leading to minimal stress for register assignment.

PROOF. From lemma 1 we know that $\mathrm{LATERIN}_j$ = TRUE is equivalent to "all paths reaching j carry placeability". $\mathrm{INSERT}_{i,j}$ = TRUE implies $\mathrm{LATERIN}_j$ = FALSE, i.e. some path reaching j does not carry placeability. From theorem 1 we know that paths reaching j and not carrying placeability must carry availability. Insertions later than in edge (i, j) would then be partially redundant. $\square$

THEOREM 4.

Let e be an expression and f a sub-expression of e. Then $\mathrm{INSERT}_{i,j}(e) \Rightarrow \mathrm{AVOUT}_i(f) + \mathrm{INSERT}_{i,j}(f)$.

Theorem 4 states that at places where an expression e is inserted all its sub-expressions are either also inserted or are already available.

PROOF. We assume that at the beginning, together with the computation of e also all its sub-expressions are computed. Further, operands of f are also operands of e. Thus, when some operands of f are modified then also some operands of e are modified. From this we derive the following implications (see also [3]):

$$\begin{aligned}
\mathrm{COMP}_i(e) &\Rightarrow \mathrm{COMP}_i(f)\\
\mathrm{TRANSP}_i(e) &\Rightarrow \mathrm{TRANSP}_i(f)\\
\mathrm{ANTLOC}_i(e) &\Rightarrow \mathrm{ANTLOC}_i(f)\\
\mathrm{AVOUT}_i(e) &\Rightarrow \mathrm{AVOUT}_i(f)\\
\mathrm{ANTIN}_i(e) &\Rightarrow \mathrm{ANTIN}_i(f)
\end{aligned}$$

Further, a path carrying availability for e also carries availability for f.

From the equations of $\mathrm{EARLIEST}_{i,j}$ we conclude

$$\begin{aligned}
\mathrm{EARLIEST}_{i,j}(e) &\Rightarrow \mathrm{EARLIEST}_{i,j}(f) + \mathrm{AVOUT}_i(f), && \text{if } i \text{ is entry block}\\
&\Rightarrow \mathrm{EARLIEST}_{i,j}(f) + \mathrm{AVOUT}_i(f) + \mathrm{TRANSP}_i(f) \cdot \mathrm{ANTOUT}_i(f), && \text{otherwise}
\end{aligned}$$

Thus, a path carrying placeability for e also either carries placeability or availability for f.

$\mathrm{INSERT}_{i,j}(e)$ = TRUE implies that all paths reaching the exit of i carry placeability for e and there is some path reaching j carrying availability for e. Thus, all paths reaching the exit of i also carry either placeability or availability for f and there is some path reaching j carrying availability for f. Hence, after all insertions have been done, f has either also been inserted in (i, j) or has become available at exit of i.

$\square$

5. PRACTICAL RESULTS

With the new code motion method presented in this paper the example from [2] have been recomputed. Figure 1 shows the result of the new algorithm applied to the example given in [2]. As explained in [2], methods based on the algorithm of Morel and Renvoise would move the computation from block 7 to the end of block 4 (redundant hoisting through a loop).

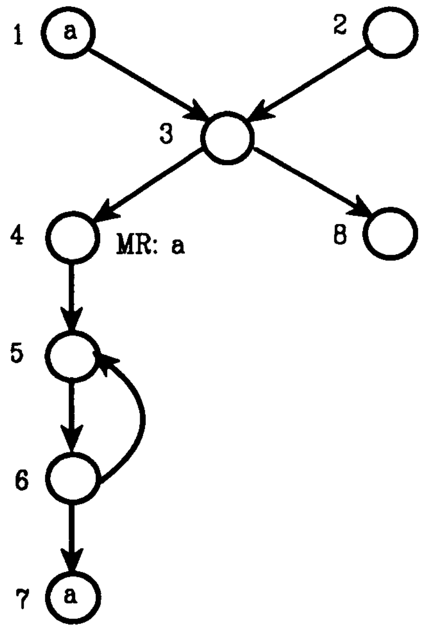

Fig. 1: Preventing redundant hoisting through a loop

Also the method presented in this paper deliveres $\text{EARLIEST}_{3,4}$ = TRUE. But then we have $\text{LATER}_{3,4} = \text{LATER}_{4,5} = \text{LATER}_{5,6} = \text{LATER}_{6,5} = \text{LATER}_{6,7}$ = TRUE and thus the computation in block 7 is not hoisted.

Next we are discussing a real world example. Figure 2 shows the flow graph of the routine xy_round in Knuth's METAFONT as it appeared during code generation for the Siemens-Nixdorf main frames (compatible to IBM 370 architecture) as target. We are concentrating our thoughts on three expressions *a, b, c* which are some compiler generated address caclulations. The original algorithm of Morel and Renvoise would insert computations of *a* and *b* at the end of node 5 and a computation of *c* at the end of node 10 (marked as "MR: ..." in Figure 2). The computations in nodes 13, 17, and 23 would then become redundant and would be deleted. The method presented here, however, inserts computations of *a* in edges (11, 17), (15, 17), (16, 17), and (18, 19) (marked as "DS: ..." in Figure 2) and deletes the redundant computations of *a* in nodes 17 and 23. The computations of *b* and *c* remain in their nodes 17 and 23 resp. This example shows that cases where the original Method of Morel and Renvoise unnecessarily increases live ranges of registers really occur in real world programs. Aside the address calculations *a, b,* and *c* there are many other computations in the basic blocks 5 to 23 which require lots of registers. Thus, it is important to keep the live ranges of the address registers *a, b,* and *c* as short as possible.

Edge Placements

Placing new computations in edges means introducing new basic blocks. This, however, requires additional jump instructions. Therefore, edge placements should be prevented as far as possible.

If an edge (i, j) is the only in-edge of its destination j, i.e. $| \mathrm{PRED}_j | = 1$, then $\mathrm{LATERIN}_j = \mathrm{LATER}_{i,j}$ and hence $\mathrm{INSERT}_{i,j} = \mathrm{FALSE}$.

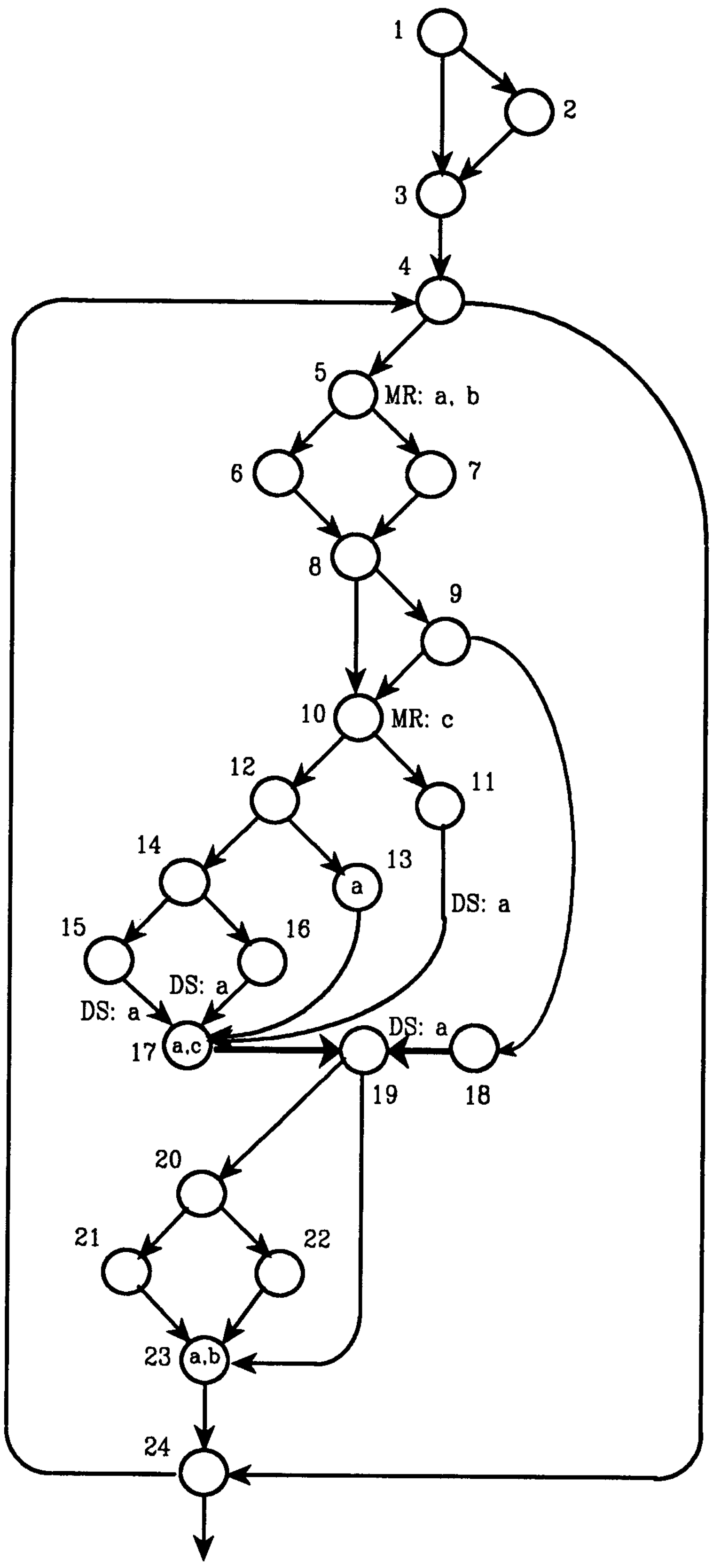

Fig. 2: Flow Graph of the Routine xy_round in Knuth's METAFONT

If $\Pi_{j \in SUCC_i} INSERT_{i,j}$ = TRUE then all the insertions can be done at the end of block *i* rather than in edges (*i*, *j*). This is always true in the case of an edge which is the only out-edge of its source *i*, i.e. $| SUCC_i | = 1$. For example, in the flow graph of Figure 2 all the insertions can be done at the end of the nodes 11, 15, 16, and 18 rather than in edges.

Thus, edge placements are at most required for *critical* edges, that are edges (*i*, *j*) with $| SUCC_i | > 1$ and also $| PRED_j | > 1$. If the code is to be generated for a pipelined processor architecture there are often *empty delay slots* at the beginning of block *j* or at end of block *i*. Computations can be inserted in such delay slots without needing extra cycles to be executed. If all computations to be inserted in an edge (*i*, *j*) fit in empty delay slots at the beginning of block *j*, then the computations should be inserted there rather than in edge (*i*, *j*). Since then their execution will need no extra time, it does not matter that they will be partly redundant. Similarly, if for all computations to be inserted in an edge (*i*, *j*) we have $ANTOUT_i$ and they all fit in delay slots at the end of block *i* (e.g. *branch delay slots*), then these computations should be inserted there rather than in edge (*i*, *j*). Again it does not matter that these computations will then be partly redundant.

6. CONCLUSION

Allowing edge placement and splitting the determination of insertions into two phases made it possible to find an improved method for elimination of partial redundancies. It could be shown that the new method is optimal, i.e. no other method is able to eliminate more partial redundancies by only safe code motions. It was further shown that live ranges of pseudo-registers are kept as short as possible, leading to minimal stress for register assignment. The presented method is also efficient; only three uni-directional systems of Boolean equations must be iteratively solved to determine insertions and deletions of computations.

With the new method elimination of partial redundancies as universal method for achieving several code optimizations at once should now have become much more attractive for practical use in compilers.

ACKNOWLEDGEMENT

The authors wish to thank their supervisors W. Frielinghaus and P. Jilek, who encouraged them to write this paper. Further they wish to thank B. Steffen for their helpful comments on an earlier version of this paper.

REFERENCES

1. Dhamdhere, D. M. A Fast Algorithm for Code Movement Optimisation. *SIGPLAN Notices* 23, 10 (1988), 172-180.
2. Dhamdhere, D. M. Practical adaption of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 291-294.
3. Drechsler, K.-H., Stadel, M. P. A solution to a problem with Morel and Renvoise's "global optimization by suppression of partial redundancies". *ACM Trans. Program. Lang. Syst.* 10, 4 (1988), 635-640.
4. Knoop, J., Rüthing, O., Steffen, B. Lazy code motion. *SIGPLAN Notices* 27, 7, (1992), 224-234.
5. Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Commun. ACM* 22, 2 (1979), 96-103, 111-126.